IMPROVING HIGH-PERFORMANCE SPARSE LIBRARIES USING COMPILER ASSISTED SPECIALIZATION: A PETSC (PORTABLE, EXTENSIBLE TOOLKIT FOR SCIENTIFIC COMPUTATION) CASE STUDY

by

Shreyas Ramalingam

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

School of Computing

The University of Utah

May 2012

The University of Utah Graduate School

STATEMENT OF THESIS APPROVAL

The thesis of **Shreyas Ramalingam**

has been approved by the following supervisory committee members:

Mary Hall	, Chair	**3/13/2012** Date Approved
Matthew Might	, Member	**3/14/2012** Date Approved
Martin Berzins	, Member	**3/15/2012** Date Approved

and by **Alan Davis** , Chair of

the Department of **School of Computing**

and by Charles A. Wight, Dean of The Graduate School.

ABSTRACT

Scientific libraries are written in a general way in anticipation of a variety of use cases that reduce optimization opportunities. Significant performance gains can be achieved by specializing library code to its *execution context*: the application in which it is invoked, the input data set used, the architectural platform and its backend compiler. Such specialization is not typically done because it is time-consuming, leads to nonportable code and requires performance-tuning expertise that application scientists may not have. Tool support for library specialization in the above context could potentially reduce the extensive understanding required while significantly improving performance, code reuse and portability. In this work, we study the performance gains achieved by specializing the sparse linear algebra functions in PETSc (Portable, Extensible Toolkit for Scientific Computation) in the context of three scientific applications on the Hopper Cray XE6 Supercomputer at NERSC.

This work takes an initial step towards automating the specialization of scientific libraries. We study the effects of the execution enviroment on sparse computations and design optimization strategies based on these effects. These strategies include novel techniques that augment well-known source-to-source transformations to significantly improve the quality of the instructions generated by the back end compiler. We use CHiLL (Composable High-Level Loop Transformation Framework) to apply source-level transformations tailored to the special needs of sparse computations. A conceptual framework is proposed where the above strategies are developed and expressed as recipes by experienced performance engineers that can be applied across execution environments. We demonstrate significant performance improvements of more than 1.8X on the library functions and overall gains of 9 to 24% on three scalable applications that use PETSc's sparse matrix capabilities.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Overview

Scientific libraries are written in a general way in anticipation of a variety of use cases that reduce optimization opportunities. Savvy application programmers are sometimes able to achieve much higher performance than library implementations of the same computation by taking the entire *execution context* into account while writing code. The process of specialization through low-level manual tuning takes significant time and expertise, and leads to nonportable, arcane code, reducing the productivity of application scientists. The process of context-specific manual library tuning can be burdensome on the application programmer who must understand not only the application and its algorithms, but also the library implementations, the architecture, the compiler and run-time mapping of the software to the architecture. The steep learning curve makes specialization an unfavorable option to improve performance.

Tool support for specializing libraries for a specific application context could combine the best of both worlds: high performance through optimizations tailored to the execution context and software reuse through libraries. This work takes an important step towards this goal, using a semi-automated and systematic process for generating specialized source code through compiler transformations. We focus our optimization system on a specific library, *PETSc (Portable, Extensible, Toolkit for Scientific Computation)*, used by more than 200 high-end applications [3]. PETSc contains high-level PDE solvers that call lower level supporting operations like BLAS functions for both dense and sparse matrices. We specialize the library in the context of three applications: PFLOTRAN, Uintah and UNIC. We demonstrate significant performance improvements of more than 1.8X on the library functions and overall gains of 9 to 24% on three scalable applications that use PETScs sparse matrix capabilities.

1.2 Motivation

Large-scale scientific applications are deployed over a large number of processors and their lifetime spans into years. For example, Uintah (discussed in this thesis) has been producing useful results for more than 10 years and uses about 40,000 production hours per year. These applications use both high-level and low-level scientific libraries to improve performance and productivity. The performance of these scientific libraries depends on the entire execution context: application, input data, algorithms, data structures, processor architecture, back end compiler, operating system etc.

Specialization is the process of optimizing code specific to frequent use cases (problem size, matrix structure etc.). Specialization has been a successful technique in improving the performance of production code. This requires careful analysis and knowledge about the software architecture. A programmer must be able to answer these questions before specializing.

- What library functions must be specialized?
- What variables can be specialized?
- What values can the variables be specialized for?
- How is the specialized code inserted into the current software architecture?
- What is the impact of the execution environment on the specialized code?

The process requires a steep learning curve and can lead to nonportable code if not done correctly. Due to its drawbacks, context-specific tuning is only approachable by an elite group of performance programmers -"Stephanie Programmers" [2]. This makes it difficult for the the application developer (also known as the "Joe Programmer") [2], who typically would be an expert in the domain but lacks in-depth knowledge about performance, since the execution context cannot be accessed by a "Stephanie Programmer". This essential disconnect can be addressed through tools and technology that aid application programmers to leverage knowledge that is currently available only to "Stephanie Programmers". In this work, we propose the use of source-to-source compilers to reduce this gap. We envision a system where "Stephanie Programmers" encapsulate optimization strategies for specific computations as transformation recipes well understood by the source-to-source compilers. These recipes would then be shipped along with the libraries to the application developer, who can then apply these transformation recipes specifically to his/her execution environment.

1.3 Libraries for Scientific Computations

Application developers use libraries extensively to solve their problems. These high-performance libraries developed by experienced programmers reduce production time significantly. Application developers typically use two types of libraries.

1.3.1 Low-Level Libraries

BLAS is a set of routines that perform the most basic set of computations required in scientific programs. Applications spend a lot of their time on matrix operations. These computations usually consist of vector operations (BLAS-1), matrix-vector multiplication (BLAS-2) and matrix-matrix operation (BLAS-3). These libraries are provided by third parties, usually by HPC vendors, and are highly optimized kernels written at a very low level to take advantage of all the features of the architecture. All the versions share a common interface and hence, application developers can link to a version they prefer, improving portability and performance of applications. BLAS libraries are highly tuned in the context of the architecture.

1.3.2 High-Level Libraries

Application developers typically use optimized higher-level scientific libraries to improve productivity and performance. Libraries like PETSc and HYPRE offer a high-level abstraction to linear and nonlinear solvers. They also provide data structures for different sparse and dense representations. In this work, we focus on PETSc as it is used by more than 200 applications and has built-in support for specialization.

We specifically focus on the *sparse linear algebra support*, since automatically tuning dense linear algebra has been extensively studied by the authors and others [6, 16, 13, 4, 27, 26] and sparse linear algebra is known to achieve very low percentage of peak due to irregular memory access patterns. Prior work on tuning sparse linear algebra has focused on a few key aspects of their implementation, including capitalizing on matrix structure, optimizing dense blocks and auto-tuning [32, 34].

1.4 Library Specialization

PETSc developers have already written an extensive set of specialized library routines for the sparse linear algebra capabilities, which provide different matrix representations, different expressions of the code, and reflect different manual optimization strategies designed to trigger appropriate responses back-end compilers. PETSc's manual specialization approach has disadvantages.

- Code is optimized for values of input parameters. These values are determined by the application developer and hence, the library developer is forced to anticipate the expected values at design time across all applications.
- Performance depends on the architecture and compiler which is unknown at design time and changes with new hardware and compiler generations.
- Library code is written in fairly low level C, including pointer arithmetic for address manipulation, which makes the code less readable and might make it difficult for a compiler to prove aliases are false, leading to loss in optimization opportunities.
- Writing many different versions of code manually is time-consuming and error-prone. The library also ends up being larger than necessary. The compiler can easily generate different versions of the code, and can only generate those that are needed by a specific application.
- Specialization with more knowledge about an application and its execution context can achieve much higher performance.

1.5 Compiler-Assisted Specialization

The focus of this work is to present a compiler-based framework that applies source-to-source optimizations on high-level sources that perform optimizations during application assembly that are much lower level than would be reasonable for portable software but lead to much higher performance. We envision this system to be used by Stephanie programmers to build libraries of PETSc *optimization strategies* that can be applied to application code once the execution context is known. Encapsulated into *transformation recipes*, these optimization strategies along with the high-level code could be shipped with the library or used directly by the application developers so that Joe programmers can automatically map the higher-level expression of the code to specialized implementations.

1.5.1 CHiLL Compiler Framework

To find the best implementation of a computation, we require a source-to-source compiler that is capable of generating different codes rapidly. To facilitate this, the framework provides a clean interface that can encapsulate all the optimizations the user wants to make. CHiLL [7] is a polyhedral loop transformation and code generation framework that applies high-level loop transformations with a script interface to describe the transformations. Polyhedral representation of loops facilitates compilers to compose complex loop transformations

in a mathematically rigorous way to ensure code correctness. CHiLL employs design features such as iteration space alignment and auxiliary loops to greatly expand the capability of a polyhedral framework. Further, its high-level script interface allows compilers or application programmers to use a common interface to describe parameterized code transformations to be applied to a computation.

1.5.2 Motivation: NEK5000

In previous work, CHiLL was used to specialize a matrix-matrix multiplication library at the *source level* specifically for matrix sizes used by NEK5000, a spectral element code from Argonne National Laboratory used to simulate a variety of applications in nuclear energy, astrophysics, ocean modeling, combustion and bio fluids [27, 26]. Nek5000 spends close to 75% of its time in dgemm calls (Matrix Multiplication). Autotuning and specialization improved the applications overall performance by 2.2x on a single node, and by 1.26X on 256 nodes of the jaguar Cray XT5 system at Oak Ridge National Laboratories. The original application used BLAS that was highly tuned for large matrices, different from the sizes used by this application. CHiLL in combination with autotuning was used to generate different implementation and selecting the best performing version. Autotuning is the process of systematically evaluating different implementations of a computation and selecting the best performing implementation. In this work, only one function was specialized for different sizes to replace BLAS. We want to extend our experience and encapsulate the methodology into a framework to optimize.

1.5.3 Specialization Approach for Libraries

Given that PETSc already incorporates specialized implementations, the compiler generated implementations can be easily integrated into an application without significant impact on its build process. Using the results of performance measurement on the target architecture, the optimization process consists of four phases:

1. Providing additional implementations of a computation to exploit matrix structure, improve data layout or simplify indirect array accesses common to sparse linear algebra functions

2. Source-to-source code transformation using the CHiLL polyhedral code generation and transformation framework [7].

3. Postprocessing to guide architecture-specific optimizations such as prefetching and SIMD parallelism in the multimedia extensions.

4. Autotuning to explore a collection of parameterized implementation variants and identify one that is best-suited for the current execution context.

1.6 Contributions

The work in the thesis is an initial step towards automating the generation of specialized libraries for scientific computing using compiler technology. The contributions of the work are as follows.

1. Approach: We outline an approach to specialize sparse computations in general and for PETSc in particular. The approach is implemented for three applications using different sparse representations. The generality of the approach has not been examined but we believe that it can be extended to other functions and libraries.

2. Case studies: A study on the effect of the execution environment on well-known transformations on PETSc functions is presented. We show that strategies depend on the back end compiler and introduce pre- and postprocessing optimizations that significantly improve the quality of the instructions generated. We also discuss some of the challenges in inserting specialized code into PETSc and the limitations in specialization.

3. Framework: We propose a framework that encapsulates the transformations, pre- and postprocessing strategies into recipes. These function-specific recipes can then be used in other execution environments to generate specialized computations.

CHAPTER 2

SPARSE LINEAR ALGEBRA: PETSC CASE STUDY

2.1 Introduction

In this chapter, we briefly introduce PETSc and different sparse matrix representations that the library provides. We also discuss briefly the different applications used in the case study for which PETSc is specialized. We finally present an analysis of the application and the PETSc functions they invoke.

2.2 PETSc Library

The Portable, Extensible Toolkit for Scientific Computation (PETSc) is a suite of matrix representations and routines for the scalable solution of scientific applications modeled by partial differential equations [3]. The PDE solvers employed support dense and sparse linear algebra functions, and the sparse functions are the topic of this work. The library is used by more than 200 applications and provides built-in support for specialization.

2.2.1 Sparse Linear Algebra

Sparse linear algebra relies on compact representations of a sparse matrix that, to the extent possible, only store the nonzero elements of the matrix. Auxiliary data structures are used to determine the rows and columns corresponding to the nonzero elements. In general, the more structure that can be exploited in the sparse matrix without significantly increasing computation, the better the performance of the code. Therefore, as with other sparse linear algebra libraries, PETSc supports a number of different sparse matrix representations that may be more appropriate for particular sparse matrix inputs.

- *AIJ:* The default sparse matrix representation is the common Compressed Sparse Row (CSR), typically used for unstructured sparse matrices. The matrix representation consists of a vector of 4, another vector of the same length that provides the column

associated with the nonzero and an additional auxiliary vector giving the index of the first element of each row.

- *BAIJ:* Blocked sparse matrices, as depicted in Figure 2.1,contain dense square blocks of fixed size that include all nonzero elements, and the blocks are padded where necessary with zero values. Using a dense block, the interior computations can use dense BLAS libraries or benefit from much better compiler optimization results on dense codes.

- *SBAIJ:* Symmetric blocked sparse matrices are similar to BAIJ, but for symmetric matrices.

- *MAIJ:* This matrix representation is used for restriction and interpolation operations for multicomponent problems, interpolating or restricting each component the same way independently.

For these four different matrix representations, PETSc provides library functions specific to them and common libraries such as sparse matrix-vector multiplication. In this work, we focus on the sequential implementations of these library functions that are invoked within the context of MPI applications. We optimize this code for three applications that use the first three of the four matrix representations,*AIJ*, *BAIJ*, and *SBAIJ*.

For this work, it is important to see how much specialization is already used in PETSc (version 3.0), as shown in Table 2.1.The first column shows the four representations, the second is the number of distinct functions (linear algebra and solvers) that were specialized and the third is the total number of provided implementations across all these functions,

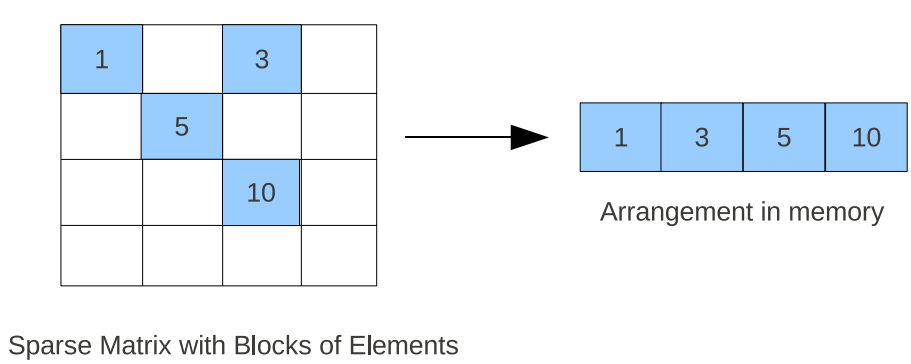

Figure 2.1: Representation of Blocked Sparse Matrices in Physical Memory

Table 2.1: Specialization in PETSc

Matrix Type	No of Specialized Functions	Total Number of Manually-Written Functions
AIJ	0	0
BAIJ	15	115
MAIJ	4	52
SBAIJ	10	75
Total	29	242

indicating the degree of specialization.While AIJ has just one implementation of each of its functions, each of the other matrix representations has several specialized implementations for each function. There are 20 different implementations of sparse matrix multiply; both BAIJ and SBAIJ have implementations specialized for specific small block sizes and a default implementation when the block is unknown, and MAIJ has 13 distinct implementations. Overall, there are 29 distinct functions specialized in PETSc described by a total of 242 implementations, an almost *order of magnitude* increase in code size due to the specialization employed manually by PETSc developers. In the remainder of this document, we will describe a system that can reduce this specialized code to a number of implementations that is much closer to 29 than 242 and also capable of achieving significant performance gains.

2.3 Applications for this Study

2.3.1 PFLOTRAN

PFLOTRAN is a highly scalable subsurface simulation code that solves multiphase groundwater flow and multicomponent reactive transport in three-dimensional porous media, and is used to study the effects of geological sequestration of CO2 in deep reservoirs and migration of other environmental contaminants in ground water [19]. PFLOTRAN spends about 30% of its time in PETSc routines. It uses a *BAIJ* block sparse matrix representation, and the block size is fixed to 15 throughout the application.

2.3.2 Uintah

The Uintah Problem Solving Framework (Uintah) was designed to provide a general framework in which a wide variety of large scale, massively-parallel simulations can be conducted [9]. The specific problem that has driven its creation is the modeling of the interactions between hydrocarbon fires, structures and high energy materials (explosives and

propellants). In this work, we consider a specific application developed using Uintah called MPMARCHES, a finite-volume large eddy simulation code used to predict the heat-flux from large buoyant pool fires with potential hazards immersed in or near a pool fire of transportation fuel. It couples a Material Point Method (MPM) description of a solid object to include stationary solids with and without conjugate heat transfer [11]. MPM is a particle method that Uintah uses particles to represent solids and the arches fluid flow solver for liquids and gasses. Uintah uses the *AIJ* matrix representation.

2.3.3 UNIC

UNIC is a 3D unstructured deterministic neutron transport code that solves a second-order form of transport using FEM (PN2ND and SN2ND) and a first-order form by method of characteristics [17]. The neutron transport code enables researchers to obtain a highly detailed description of a nuclear reactor core.The application spends more than 50% of its time on PETSc routines. It uses an *SBAIJ* matrix representation, but with a block size of 1, meaning that it is really just a symmetric *AIJ* matrix.

2.4 Performance Analysis

The results of performance analysis for the three applications are given in Table 2.2. These results were obtained by running HPCToolkit [1] on a single node of hopper, a Cray XE6 system at NERSC. In the table, each application is listed in the first column. The second column gives the PETSc function names for the key computations in the application, and the third column is the percentage of overall execution time. The function names incorporate the matrix representation and the function to be applied. For Uintah, the PETSc functions in parentheses invoke the lower-level routines. All three applications spend significant time in sparse matrix multiplication, along with additional routines. We focus our study on five out of the seven routines in Table 2.2. We omit the sparse matrix multiplication invoked by *ApplyFilter* because of its modest impact on execution time and the need for a different specialization than the other implementation. We also omit the *LU Factorization* from consideration in PFLOTRAN, since a previous study of this code on Jaguar, a Cray XT5, using PAPI, determined that it was already achieving 20% of peak performance, while the other two functions were performing below 5% of peak [28].[1]

[1]Access to PAPI is not provided by HPCToolkit on hopper.

Table 2.2: Application Performance Analysis

Application	PETSc Function	% Exec. Time
PFLOTRAN	MatLUFactorNumeric_SeqBAIJ_N MatSolve_SeqBAIJ_N MatMult_SeqBAIJ_N	10% 9.8% 9.8%
Uintah	MatMult_SeqAIJ (PetscSolve) MatMult_SeqAIJ (ApplyFilter)	23% 3%
UNIC	MatMult_SeqSBAIJ_1 MatRelax_SeqBAIJ	39.7 % 46.9%

2.5 PETSc Functions

In this section, we outline the codes optimized in Table 2.2 and briefly explain their behavior.

2.5.1 Matrix-Vector Multiplication

All three applications invoke Matrix-Vector Multiplication and the code for the three version of matrix types (AIJ,BAIJ and SBAIJ) is shown in Figure 2.2, Figure 2.3 and Figure 2.4. In each function, variable i iterates over the rows (AIJ) or blocked rows [2] (BAIJ,SBAIJ) and calculates the number of nonzero elements (variable n) at each iteration. Once calculated, the codes perform vector dot product for each row for their respective matrix types. The code for BAIJ differs a bit from the code displayed here. The original code performs a gather operation (discussed in the next section) and uses BLAS to perform Matrix-Vector multiplication for each block. The code for SBAIJ is for block size of 1 and performs operations for matrix data on both sides of the diagonal.

[2]Blocked Row: The number of rows in a Blocked Row is equal to the size of the block

```
for(i=0;i<m;i++){
   /* Calculate Number of Non Zero Elements */
   n=ii[i+1] - ii[i];
   y[i]=0.0;
   /* Vector Product - Column Major */
   for(j=0;j<n;j++)
  y[i]+= aa[ii[i]+j]*x[aj[ii[i]+j]];

}
```

Figure 2.2: AIJ- Matrix-Vector Multiplication

```
for(i=0;i<m;i++)
    //Initialization of Pointers

   /* Calculate Number of Non Zero Blocks */
     n    = ai[1] - ai[0];
    for(k=0;k<bs;k++)
  z[k]=0.0;
  /* Matrix-Vector Multiplication for Blocked Row */
  /* Row Major */
    for (k=0; k<n; k++){
  xb = x + bs*(*idx++);
  for(l=0;l<bs;l++)
    for(j=0;j<bs;j++)
       z[l]+= v[l*bs+j]*xb[j];
   v+=bs*bs;
   z+=bs;
     }
```

Figure 2.3: BAIJ - Matrix-Vector Multiplication

```
for (i=0; i<mbs; i++) {
     //Initialization of Pointers

     /* Calculate Number of Non Zero Blocks */
     n    = ai[1] - ai[0];

     /* Vector Dot Product for Row */
     for (j=jmin; j<n; j++) {
       cval    = *ib;
       z[cval] += *v * x1;
       z[i]    += *v++ * x[*ib++];
     }
   }
```

Figure 2.4: SBAIJ: Matrix-Vector Multiplication

2.5.2 Other Functions

The other two functions are MatSolve for PFLOTRAN and MatRelax in UNIC. Both functions perform a forward sweep and then a reverse sweep. The core computation is very similar to that of Matrix-Vector multiplication but performs a subtraction. The code is outlined in Figures 2.5 and 2.6. At each iteration i, the number of nonzeros are calculated and the matrix operations are performed. Just as in the case of MatMult in BAIJ, the innermost (l,j) loops are replaced by a BLAS call in the real code but no gather operation is performed.

```
   /* Forward solve the upper triangular */
   for (i=1; i<n; i++) {

    /* Calculate Number of Non Zero Blocks */
     n = ai[i+1] - ai[i];
     //Initialization of pointers
     for(k=0;k<n;k++){
  double* ptr=t+bs*vi[k];
  for(l=0;l<bs;l++)
    for(j=0;j<bs;j++)
    s[j] -= v[i*bs+j]*ptr[i];
      v += bs2;
     }
   }

   /* backward solve the upper triangular */
   ls = a->solve_work + A->cmap->n;
   for (i=n-1; i>=0; i--){

    /* Calculate Number of Non Zero Blocks */
     n = adiag[i] - adiag[i+1]-1;
     //Initialization of pointers
     for(k=0;k<n;k++){
  double* ptr=t+bs*vi[k];
  for(l=0;l<bs;l++)
    for(j=0;j<bs;j++)
    ls[j] -= v[i*bs+j]*ptr[i];
     v += bs2;
    }
     Kernel_w_gets_A_times_v(bs,ls,aa+bs2*adiag[i],t+i*bs);
          /* *inv(diagonal[i]) */
     PetscMemcpy(x+i*bs,t+i*bs,bs*sizeof(PetscScalar));
   }
```

Figure 2.5: BAIJ: MatSolve

```
for (i=0; i<m-1; i++){ /* update rhs */
    //Initialization of Pointers

    /* Calculate Number of Non Zero Blocks */
    n = ai[i+1] - ai[i] - 1;
    while (n--) t[*vj++] -= x[i]*(*v++);
  }
  for (i=m-1; i>=0; i--){
    //Initialization of Pointers

    /* Calculate Number of Non Zero Blocks */
     n = ai[i+1] - ai[i] - 1;
     sum = t[i];
     while (n--) sum -= x[*vj++]*(*v++);
     x[i] =  (1-omega)*x[i] + omega*sum/d;
}
```

Figure 2.6: SBAIJ: Matrix Relax

2.6 Summary

In this chapter, we discussed the different matrix types and their associated specializations supported by PETSc. We also discussed in brief the applications and the PETSc functions they invoke. It is important to note that a majority of sparse computations need to dynamically calculate the number of nonzero elements for each unit row. This is one of the variables apart from matrix properties (block size) that we employ for specialization.

CHAPTER 3

OPTIMIZATION APPROACH

3.1 Introduction

In this chapter, we give a brief overview of the optimization approach used in this work to gain performance. We discuss some of the challenges in optimizing sparse computations and techniques that we used to overcome these challenges. Finally, we study the functions and their specializations in more details for the applications in this section.

3.2 Overview

In our optimization of the three applications, we used the following general approach depicted in Figure 3.1.

1. *Profiling:* Since the performance of the sparse codes is heavily dependent on matrix structure, a profiling pass needs to collect dynamic information about the PETSc functions within the application and overall execution context. This pass identifies PETSc functions that comprise a significant fraction of the application's execution time. It also measures the number of nonzeros per row of the matrices invoked by these functions, to determine whether some matrix structure can be exploited. This profiling information is described in Section 3.4.1.

2. *Code variants:* For each application, we provide an additional implementation of its main computations to exploit matrix structure and perform *gathers* of sparse data to improve data layout (in memory or registers) for the indirect array accesses common to sparse linear algebra functions, as described in Section 3.4.2. The code variants are specific to the functions of PETSc.

3. *Source-to-source code transformation:* We use the CHiLL polyhedral code generation and transformation framework to perform the code transformations in Table 3.1 and described in Section 3.4.3 that are useful for sparse matrix computations. CHiLL may also generate specialized variants of a computation, and these are integrated into the

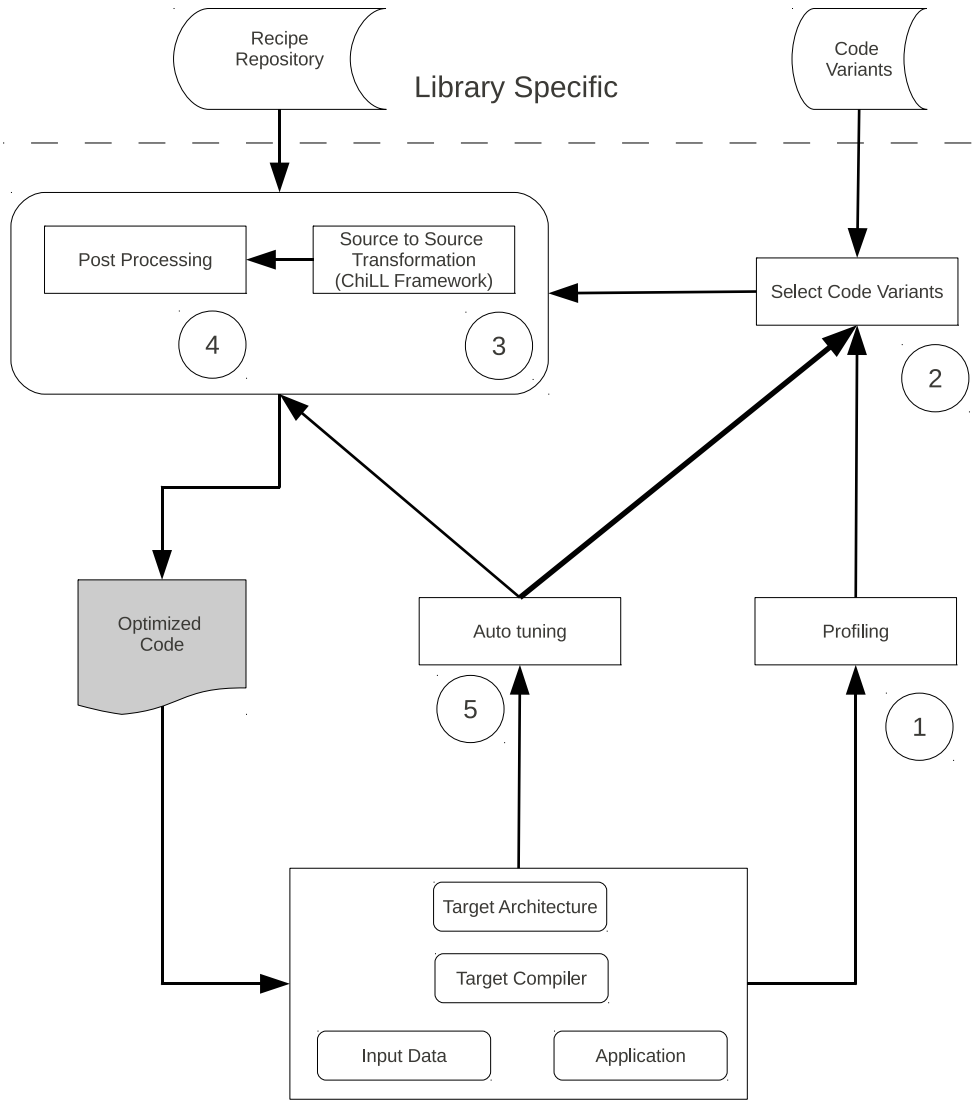

Figure 3.1: Compiler-assisted approach to specialization

application implementation.These transformations are expressed with transformation recipes, which are maintained in a repository with the application (or possibly PETSc itself).

4. *Architecture-specific postprocessing:* We apply architecture-specific pragmas and additional modifications in a post-processing stage, as described in Table 3.1 and discussed in Section 3.5.4.

5. *Autotuning:* In conjunction with CHiLL's code generation, we employ autotuning to search the possible implementations of a computation and identify the best-performing

Table 3.1: Transformations and pragmas for optimizing sparse matrix code.

Name	What it does	Benefit to sparse code
CHiLL Transformations and Data Specialization		
unroll	applies unroll and unroll-and-jam to inner two loops	exposes ILP and SIMD parallelism, register reuse, and reduces branches
split	splits the iteration space of loops into separate loops	exposes opportunities for SIMD parallelism
distribute	distributes a loop across statements in its body	permits finer control of statement ordering
fusion	fuses multiple loops into a single loop	permits finer control of statement ordering
known	adds integer constraints to loop iteration spaces	provides bounds on variable values to be used in specialization
Post-processing		
prefetch	load data into cache prior to its use	library call inserted into the code
vector always	pragma for SSE SIMD code	force Intel compiler to generate SIMD SSE code
scalar replacement	copy array variables into scalars	force PGI compiler to use more registers

solution in the context of the application, as described in Section 3.5.5.

3.3 Optimizing Sparse Code

Several properties of sparse codes impact their performance and make it challenging for the backend compiler to generate high-quality code.

- *Unknown loop bounds:* When the loop bounds are unknown, specializing according to loop bounds is not possible. It also makes the decision as to whether it is profitable to use SIMD instructions difficult for the backend compiler. Therefore, when possible we employ dynamic data to determine loop bounds.

- *Small loop bounds:* Even when loop bounds are known, if they are small as is usually the case with inner loops iterating over nonzeros in a row, the backend compiler is often overly conservative in employing SIMD SSE instructions.

- *Indirect indexing expressions:* Indirect indexing expressions present a number of challenges to compilers. Compiler transformations are most successful in the affine domain, where loop bounds and subscript expressions are linear functions of the loop indices. Indirect accesses are not affine, but some optimizations can still be performed as long as there are no dependences. Additional optimizations may be possible by

reducing the indirection or moving it to a different position in the code (*i.e.,* a gather discussed below).

- *Limited data reuse and impact on memory bandwidth:* In sparse code, there is very little reuse of data in cache. The number of floating point operations performed per memory load is relatively low and therefore, memory bandwidth becomes a key limiting factor to performance.

Compiler optimizations for sparse codes must therefore be very different than for dense codes. For large dense matrices, for example $1024 - by - 1024$, dense linear algebra libraries can achieve close to peak performance. They incorporate aggressive memory hierarchy optimizations such as *data copy*, *tiling* and *prefetching* to reduce memory traffic and hide memory latency. Additional code transformations improve *instruction-level parallelism* (ILP). Several examples describe this general approach [6, 33, 8, 12, 36].

In previous work, we showed that dense BLAS libraries do not perform well for small dense matrices [27]. Since these matrices fit within even small L1 caches, the focus of optimization should be on managing registers, exploiting ILP in its various forms and reducing loop overhead. For these purposes,we can use *loop permutation* and aggressive *loop unrolling* for all loops in a nest. To the backend compiler, unrolling exposes opportunities for instruction scheduling, scalar replacement, and eliminating redundant computations. Loop permutation may enable the backend compiler to generate more efficient *single-instruction multiple-data* (SIMD) instructions by bringing a loop with unit stride access in memory to the innermost position, as required for utilization of multimedia-extension instruction set architectures. This approach was used to specialize a dense BLAS library for *nek5000*.

In considering how to optimize the sparse functions in PETSc, we need to perform these and additional optimizations that were not needed for the small dense codes. We still must perform aggressive loop unrolling to expose instruction-level-parallelism and register reuse. Due to the chosen loop order in PETSc and differences in sparse code, loop permutation is not useful. Specializing code for particular matrix sizes is much more difficult, as the number of nonzeros is often nonuniform, but we will describe how to do this with dynamic data. Exploiting SIMD code generation is far more difficult due to the small loop bounds, which the backend compiler may determine are too small to be profitable. In addition, for loop iteration counts that are irregular or simply just odd, the backend compiler avoids SSE instructions due to concerns about alignment to boundaries. Finally, to recognize SSE instructions, the compiler may look for code with a certain structure (*e.g.,* a dependence-free statement in a loop with unit stride). Therefore, *loop splitting* in combination with pragmas

is used to force SSE code generation when profitable. To preload data into cache and improve memory latency and bandwidth, we include *prefetch* instructions. This set of transformations is described in Table 3.1.

3.4 Optimizing the Applications

We describe the optimization approach of the previous section using concrete examples from the three applications.

3.4.1 Profiling

In addition to gathering the execution frequency data previously shown in Table 2.2, a profiling phase must also derive the frequency of numbers of nonzeros per row for the matrices accessed by the PETSc routines. We show this information in Tables 3.2, 3.3 and 3.4, and we will use it subsequently to optimize the code.

3.4.2 Code Variants

We describe the code variants used in our experiment, which improve known performance issues with sparse code and could be integrated into PETSc itself and used by other applications.

- *Gather Operations (Memory Level):* We modified the functions to use two very different gather operations, in PFLOTRAN and UNIC. The gather in PFLOTRAN was actually present in some but not all of the specialized PETSc routines provided by the PETSc developers, as previously described by Table 2.1. Recall that PFLOTRAN

Table 3.2: PFLOTRAN: nonzero blocks per block row.

MatMult_SeqBAIJ_N		MatSolve_SeqBAIJ_N	
n	frequency	n	frequency
5	4% to 8%	1	0% to 1%
6	30% to 45%	2	15% to 20%
7	50% to 75%	3	75% to 85%

Table 3.3: Uintah: nonzero elements per row

n	Frequency
7	85.7%
6	13.5%
5	0.7%

Table 3.4: UNIC: Nonzero elements per row.

n	Frequency
27	12%
25	10%
18	10%
49	8.5%
28	4.3%
32	3.7%
14	3.7%

uses the BAIJ blocked sparse matrix representation, as shown in Figure 2.1. In the code without gather, for each blocked row, the code iterates through each block, multiplying the block with its corresponding data are the vector. The vector data are therefore accessed through indirection, which may lead to poor memory performance, as shown in Figure 3.2. Some *BAIJ* functions in PETSc instead perform a gather operation which collects the vector data for each block into a contiguous array. The array can then be multiplied with the blocked row as a whole, as shown in Figure 3.3. In the gather version, the code accesses the vector from a contiguous array and can then invoke a BLAS library to perform a Matrix-Vector multiplication for a rectangular block. While the quality of the code for the multiplication is going to be far superior to the original code that includes indirection, the overhead of performing the gather means that it is not always profitable, and therefore should be evaluated in the application context. By having two variants, one with gather and one without, we can evaluate the benefit in conjunction with the other compiler optimizations.

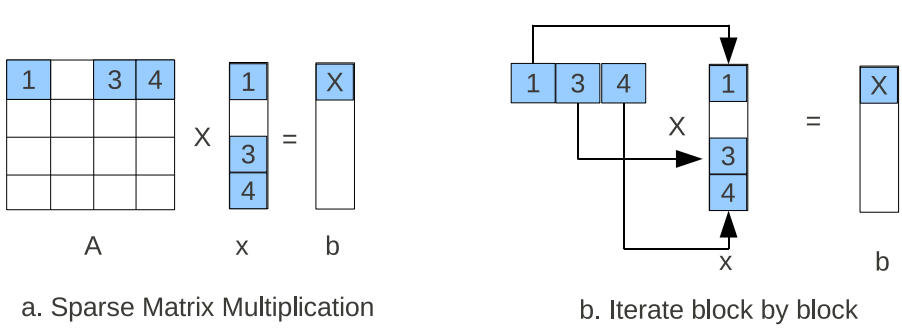

Figure 3.2: Sparse Block Multiplication (Non-Gather Operation)

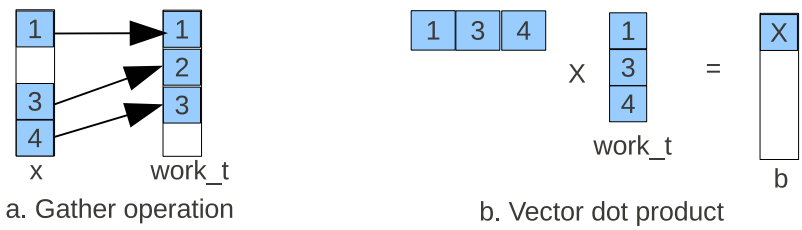

Figure 3.3: Sparse Block Multiplication (Gather Operation)

- *Gather Operation (Register Level):* While this gather is used to improve data layout in memory, we used another gather operation in the *AIJ* MatMult in Uintah to perform a gather in a register and expose opportunities for increased compiler optimization. The code for *AIJ* MatMult (modified for the ELL format below) is shown in Figure 3.4, and the code that performs the gather is found in Figure 3.5. The benefit of rewriting the code in this way is that it separates the non-affine array index expression `aj[i*n+j]` from the array access. The modified code gathers the values of `x[aj[i*n+j]]` into temporary array `colVal[j]`. The gather version, in conjunction with the optimizations below, can improve the mapping of data to registers, and for sufficiently large loop bounds, can expose opportunities for SIMD SSE instructions. A similar gather modification is included in the *SBAIJ* code for UNIC.

- *Matrix Representation:* Uintah uses the *AIJ* matrix representation, which is designed for unstructured sparse matrices. However, it does in fact have a structure we can exploit; it is a *diagonal banded matrix*, meaning that all nonzeros fall on a band around the diagonal. Such matrices (Figure 3.6) can be viewed as descriptions of the coupling between the problem variables. The bandedness corresponds to the fact that variables

```
for(i=0;i<m;i++){
   for(j=0;j<n;j++)
  y[i]+= aa[i*n+j]*x[aj[i*n+j]];

}
```

Figure 3.4: MatMult_SeqAIJ- ELL

```
for(i=0;i<m;i++){
   //Load into temporary arrays

   for(j=0;j<n;j++){
  col[j] = aj[i*n+j];
  colVal[j] = x[col[j]];
  y[i]+= aa[i*n+j]*colVal[j];
  }

}
```

Figure 3.5: MatMult_SeqAIJ- ELL with Gather

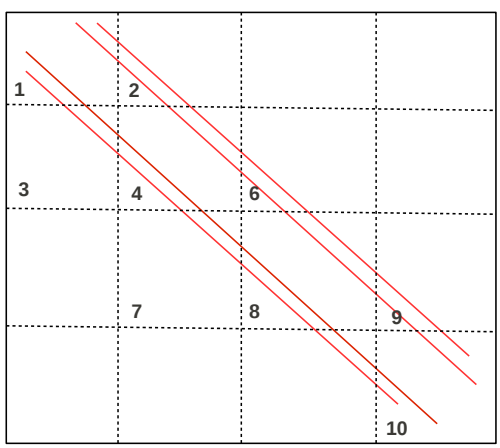

Figure 3.6: Banded Matrices in PETSc

are not coupled over arbitrarily large distances. As a result of profiling, from Table 3.3 we see that the number of nonzeros per row is fairly uniform and small (no more than 7), and we can use a fixed number of nonzeros $nz = 7$. A sparse matrix with a fixed number of elements per row is known as an ELL representation [24]. Since we are minimizing impact to the existing code, we simulate an ELL representation in the *AIJ* code by simply padding with zeros any rows in the *AIJ* that have fewer than nz elements. Once the row size is fixed, the compiler can specialize the code for that size, greatly improving the performance.

3.4.3 Code Transformations in CHiLL

The above code variants, to gather elements of sparse vectors and exploit the banded matrix structure, set the stage for further specialization in the compiler. For all three applications, we optimize the code in the same way: *known* provides an interface to specialize code according to fixed loop bounds and block sizes, *unroll* at multiple loop levels to expose instruction-level parallelism, SSE code generation and simplify branching, and in some cases, *split* the iteration space of the loop prior to inserting pragmas to force SSE code generation, discussed in the next section. CHiLL's optimization and code generation strategy can be controlled using a *transformation recipe* that describes the set of transformations to be applied [14]. Whether to split the loop and which loops to unroll can be represented with different CHiLL recipes. Which of these recipes to use and how much to unroll is determined through 1.

3.4.4 Original vs Gather (Memory Level)

We performed some tests on Matrix-Vector Multiplication (BAIJ) using BLAS. We set the number of nonzero blocks to 7 and each block is separated from the previous block by 200 blocks. The results are plotted in Figure 3.7. Gather performs significantly better than the original for small block sizes. However, for block sizes more than 20, the original code performs better. This is probably because BLAS is column-major and performs very

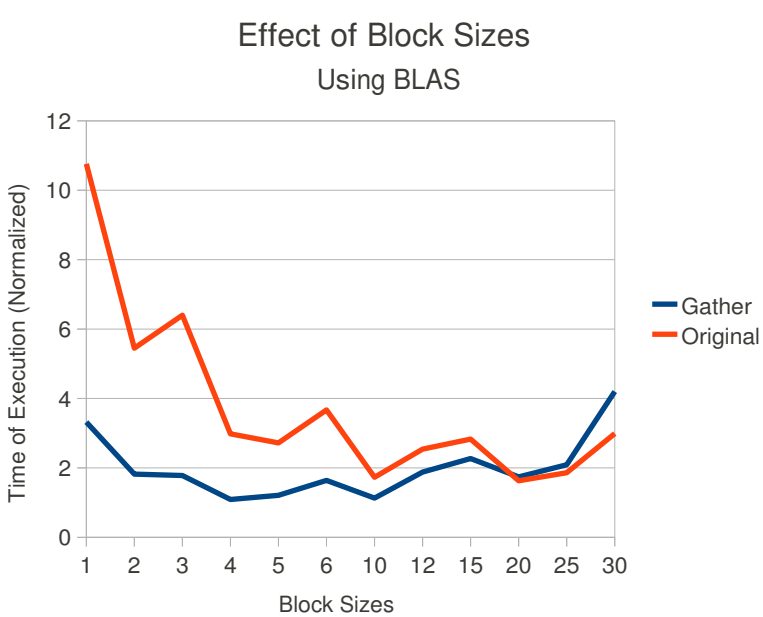

Figure 3.7: Original vs Gather : Varying Block Sizes

well for small number of rows and large number of columns. For small block sizes, gather causes the number of columns to increase but the number of rows remain constant. As the size of the blocks increase, the row dimensions increase, reducing performance. The shift in performance occurs between block sizes 15 to 20 and depends on the distance between the blocks. From our experiments, the effect of distance only effects the block size at which the shift in performance occurs. The effect of distance for a a constant number of blocks (n=7) and Block Size (bs=15) is displayed in Figure 3.8. Note that these results use BLAS and might vary depending on the code used to perform Matrix-Vector Multiplication.

3.5 Examples

Below are three examples of scripts and generated, one from each application.

3.5.1 Example 1

The example script shown in Figure 3.9 was used for the MatMult (BAIJ) in pflotran. Recall that block sizes of the matrices were 15. The code is outlined in Figure 2.3 in Chapter 2. The inner two loops were optimized where the inner loop was always bounded by 15 and the outer loop is bounded by 15 or 15*n (gather version). The script achieves significant performance gains in the PGI compiler. It unrolls the innermost loop by a factor 5 and the outer loop by a factor of 3. The unrolled code is then postprocessed for scalar replacement and prefetching. The script and the generated code is shown in Figures 3.9 and 3.10.

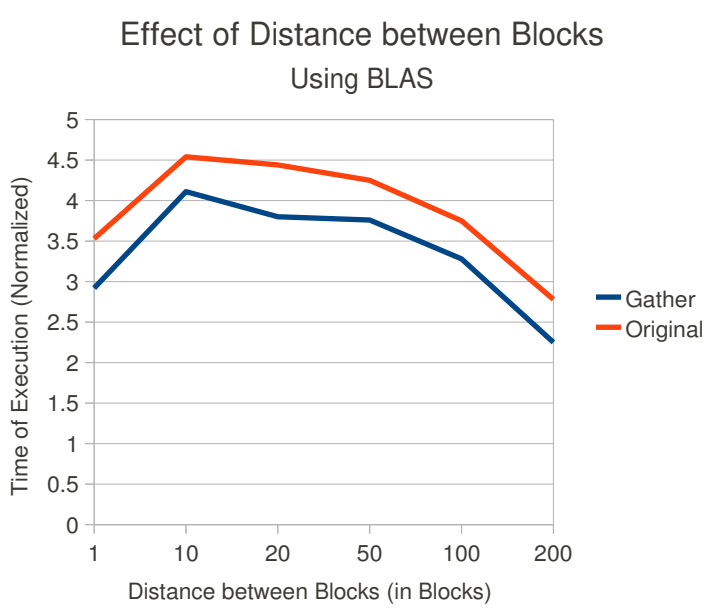

Figure 3.8: Original vs Gather : Varying Distance

```
known(bs<16)
known(bs>14)
ul_m = 5
ul_n = 3
//and unroll loops 1 and 2
//Syntax: unroll(statement no,loop no,
    unroll factor)
unroll(0,2,ul_m)
unroll(0,1,ul_n)
```

Figure 3.9: MatMult-BAIJ - CHiLL Transformation Recipe

```
for (t4 = 0; t4 <= 12; t4 += 3)
 {
//Post Processing - Prefetch A for 5 iterations ahead for 3
    rows
   _mm_prefetch((char*)&A[(t4*15)+40], _MM_HINT_T0);
   _mm_prefetch((char*)&A[(t4*15)+48], _MM_HINT_T0);
   _mm_prefetch((char*)&A[(t4*15)+56], _MM_HINT_T0);
   _mm_prefetch((char*)&A[(t4*15)+64], _MM_HINT_T0);
   _mm_prefetch((char*)&A[(t4*15)+72], _MM_HINT_T0);
   _mm_prefetch((char*)&A[(t4*15)+80], _MM_HINT_T0);

    //Post Processing - Scalar replacements
    chillTemp_5 = x[t4];
    chillTemp_7 = x[t4 + 2];
    chillTemp_6 = x[t4 + 1];
    for (t6 = 0; t6 <= 10; t6 += 5)
    {
     chillTemp_0 = z[t6];
     <-- copy z array values to chillTemp_ scalars -->
     chillTemp_0 += A[t4 * 15 + t6] * chillTemp_5;
     chillTemp_1 += A[t4 * 15 + t6 + 1] * chillTemp_5;
     chillTemp_2 += A[t4 * 15 + t6 + 2] * chillTemp_5;
     chillTemp_3 += A[t4 * 15 + t6 + 3] * chillTemp_5;
     chillTemp_4 += A[t4 * 15 + t6 + 4] * chillTemp_5;
     chillTemp_0 += A[(t4+1) * 15 + t6] * chillTemp_6;
     chillTemp_1 += A[(t4+1) * 15 + t6 + 1] * chillTemp_6;
     chillTemp_2 += A[(t4+1) * 15 + t6 + 2] * chillTemp_6;
     <--- Rest of the Code --->
     z[t6] = chillTemp_0;
     <-- copy from chillTemp_ back to z -->
     }
   }
    return;

}
```

Figure 3.10: MatMult - BAIJ - Unrolling with Post-processing (Xcalar Rep + Prefetch)

3.5.2 Example 2

An example CHiLL recipe for Uintah which employs both gather and loop splitting to expose SSE is shown in Figure 3.11. The script transforms the code shown in Figure 3.5 (ELL). All statements are indexed starting with value 0 and all loops are indexed starting at value 1. The code is specialized for the value 7 using the known command. The three statements are distributed and the first two are unrolled completely. The simple inner loop structure, split into loops with iteration counts that are small powers of two, in conjunction with the postprocessing below, forces the Intel compiler to use SSE instructions. The generated code is displayed in Figure 3.12.

3.5.3 Example 3

Another example of performing gather with loop unrolling to expose instruction-level parallelism, register reuse and possible SSE instructions is shown in Figure 3.13. We do

```
value_n= 7
known(n>value_n-1)
known(n<value_n+1)
original()
//Distribute the statements
//Syntax: distribute([statement nos,loop
      no])
distribute([0,1,2],2)

//and unroll only statement 0 and 1
//Syntax: unroll(statement no,loop no,
     unroll factor)
unroll(0,2,7)
unroll(1,2,7)

//Split the loop
//Syntax:
// split(statement no,loop no,split
     condition)
split(2,2,L2<6)
split(2,2,L2<4)
```

Figure 3.11: MatMult-AIJ - CHiLL Transformation Recipe

```
for (t2 = 0; t2 <= m - 1; t2++) {
    //Prefetch matrix aa and column vecto aj
    _mm_prefetch((char*)&aa[(t2*7)+40], _MM_HINT_T0);
    _mm_prefetch((char*)&aj[(t2*7)+80], _MM_HINT_T0);

   *col = aj[t2 * 7 + 0];
    col[0 + 1] = aj[t2 * 7 + 0 + 1];
    col[0 + 2] = aj[t2 * 7 + 0 + 2];
    col[0 + 3] = aj[t2 * 7 + 0 + 3];
    col[0 + 4] = aj[t2 * 7 + 0 + 4];
    col[0 + 5] = aj[t2 * 7 + 0 + 5];
    col[0 + 6] = aj[t2 * 7 + 0 + 6];
    *colVal = x[*col];
    colVal[0 + 1] = x[col[0 + 1]];
    colVal[0 + 2] = x[col[0 + 2]];
    colVal[0 + 3] = x[col[0 + 3]];
    colVal[0 + 4] = x[col[0 + 4]];
    colVal[0 + 5] = x[col[0 + 5]];
    colVal[0 + 6] = x[col[0 + 6]];
    #pragma vector always
    for (t4 = 0; t4 <= 3; t4++)
      y[t2] = y[t2] + aa[t2 * 7 + t4] * colVal[t4];

    #pragma vector always
    for (t4 = 4; t4 <= 5; t4++)
      y[t2] = y[t2] + aa[t2 * 7 + t4] * colVal[t4];
    y[t2] = y[t2] + aa[t2 * 7 + 6] * colVal[6];
  }
}
```

Figure 3.12: MatMult - BAIJ - Gather + loop splitting with Postprocessing (Prefetching)

not perform any post processing on this code as it degrades performance. In this we do not unroll completely as it might cause register spilling. The generated code is shown in Figure 3.14.

3.5.4 Postprocessing

We use postprocessing to insert calls to the prefetch API and pragmas specific to the Intel compiler to force SSE code generation, as in the examples above. The matrix data *aa* and column vector *aj* are prefetched 5 iterations ahead. The prefetch distance depends on the architecture while the number of prefetch statements is dependent on the amount of work done inside the loop. Pragmas that compel the compiler to generate SSE code are inserted into split loops.

These optimizations modify the code in an architecture-specific way, and are not portable source-to-source transformations. Therefore, they are not included in CHiLL and are instead added by a postprocessing phase.

```
value_n= 7
known(n>value_n-1)
known(n<value_n+1)
original()

value_n= 14
known(n>value_n-1)
known(n<value_n+1)

original()
//Distribute the statements
//Syntax: distribute([statement nos,loop
      no])
distribute([0,1,2,3],2)

//and unroll statements 0,1,2 and 3 by 6
unroll(0,2, 6)
unroll(1,2, 6)
unroll(2,2, 6)
unroll(3,2, 6)
fuse([0,1,2,3],2)
print
```

Figure 3.13: MatMult-SBAIJ - CHiLL Transformation Recipe

```
*z_temp = z[i];
for (t4 = 1; t4 <= 7; t4 += 6)
   {
     col[t4] = ib[t4];
     col[t4 + 1] = ib[t4 + 1];
     col[t4 + 2] = ib[t4 + 2];
     col[t4 + 3] = ib[t4 + 3];
     col[t4 + 4] = ib[t4 + 4];
     col[t4 + 5] = ib[t4 + 5];
     colVal[t4] = x[col[t4]];
     colVal[t4 + 1] = x[col[t4 + 1]];
     colVal[t4 + 2] = x[col[t4 + 2]];
     colVal[t4 + 3] = x[col[t4 + 3]];
     colVal[t4 + 4] = x[col[t4 + 4]];
     colVal[t4 + 5] = x[col[t4 + 5]];
     z[col[t4]] += v[t4] * x1;
     z[col[t4 + 1]] += v[t4 + 1] * x1;
     z[col[t4 + 2]] += v[t4 + 2] * x1;
     z[col[t4 + 3]] += v[t4 + 3] * x1;
     z[col[t4 + 4]] += v[t4 + 4] * x1;
     z[col[t4 + 5]] += v[t4 + 5] * x1;
     *z_temp += v[t4] * colVal[t4];
     *z_temp += v[t4+1] * colVal[t4+1];
     *z_temp += v[t4+2] * colVal[t4+2];
     *z_temp += v[t4+3] * colVal[t4+3];
     *z_temp += v[t4+4] * colVal[t4+4];
     *z_temp += v[t4+5] * colVal[t4+5];
   }
 col[13] = ib[13];
 colVal[13] = x[col[13]];
 z[col[13]] = z[col[13]] + v[13] * x1;
 *z_temp = *z_temp + v[13] * colVal[13];
 z[i] = *z_temp;
```

Figure 3.14: MatMult - SBAIJ - Gather + Unrolling + with Postprocessing (Prefetching)

3.5.5 Autotuning

Autotuning technology systematically explores a search space of alternate implementations of a computation to select the best-performing solution for a particular execution context. CHiLL's structure is designed to support its use in autotuning. For the purposes of this paper, we use autotuning to select among code variants and CHiLL transformation recipes and to fine-tune loop unrolling parameters for a particular transformation recipe applied to a particular code variant. Often a concern with autotuning is the size of the search space. In previous work, we have employed Parallel Rank Order search to navigate a compiler's optimization search space, by integrating with the Active Harmony system [29, 30]. For example, we showed in [30] we could explore a search space of over 500 million points by looking at only 490 points in 20 parallel steps. However, in this case, the loop bounds are small, and by using unroll factors divisible by the loop bounds, we can search the entire space.

3.6 Summary

In AIJ matrices, if the matrix cannot be converted to a an ELL format, the function has to be specialized for values of the number of nonzeros per row. In BAIJ and SBAIJ, code can be specialized irrespective of the number of nonzeros per row. By employing a gather operation, code can also be specialized for the number of nonzeros in addition to the block size. In UNIC, the block size is only one and hence the function is specialized for a varying number of nonzeros which results in a large piece of code, as shown in Figure 3.15. In this section, we outlined our approach to specialize PETSc functions using compiler-based source-to-source transformation in conjunction with certain pre- and postprocessing to achieve the desired results. We introduce techniques such as gather at both memory level and register level. These techniques bring about significant performance improvement as we will see in the following chapter.

```
for (i=0; i<mbs; i++) {
  //Initializations
  switch (n){
    case 27:
  //specialized code for n=27
  *z_temp = z[i];
  for (t4 = 1; t4 <= 22; t4 += 3)
  {
    col[t4] = ib[t4];
    col[t4 + 1] = ib[t4 + 1];
    col[t4 + 2] = ib[t4 + 2];
    colVal[t4] = x[col[t4]];
    colVal[t4 + 1] = x[col[t4 + 1]];
    colVal[t4 + 2] = x[col[t4 + 2]];
    z[col[t4]] += v[t4] * x1;
    z[col[t4 + 1]] += v[t4 + 1] * x1;
    z[col[t4 + 2]] += v[t4 + 2] * x1;
    *z_temp += v[t4] * colVal[t4];
    *z_temp += v[t4+1] * colVal[t4+1];
    *z_temp += v[t4+2] * colVal[t4+2];
   }
   <--- Clean up loop --->
   z[i] = *z_temp;
    break;
     case 14:
   *z_temp = z[i];
   for (t4 = 1; t4 <= 11; t4 += 2)
    {
      <--- Gather to col and colVal--->
      z[col[t4]] += v[t4] * x1;
      z[col[t4 + 1]] += v[t4+1] * x1;
      *z_temp += v[t4] * colVal[t4];
      *z_temp += v[t4 + 1] * colVal[t4+1];
     }
   <--- Clean up loop --->
    z[i] = *z_temp;
    }
    break;
<------------------------------------------>
     Specialized Code for 5 more cases
<------------------------------------------>
      default:
    for (j=jmin; j<n; j++) {
      cval   = *ib;
      z[cval] += *v * x1;
      z[i]   += *v++ * x[*ib++];
    }
   break;
  }
   }
```

Figure 3.15: Specialized Code for MatMult for SBAIJ

CHAPTER 4

RESULTS

4.1 Introduction

In this chapter, we discuss our experimental setup and workflow. We present our results for each transformation using the PGI and the Intel compilers. Finally, we discuss in detail the impact of our optimizations on the performance of the application and the impact of the back end compiler.

4.2 Experiment Workflow

Experiments were conducted in 2 phases. In phase 1, we optimize the outlined code under the assumption that data are in the cache. We use a simple driver function written in C that evaluates the code. The workflow for Phase 1 is shown in Figure 4.1. For a given

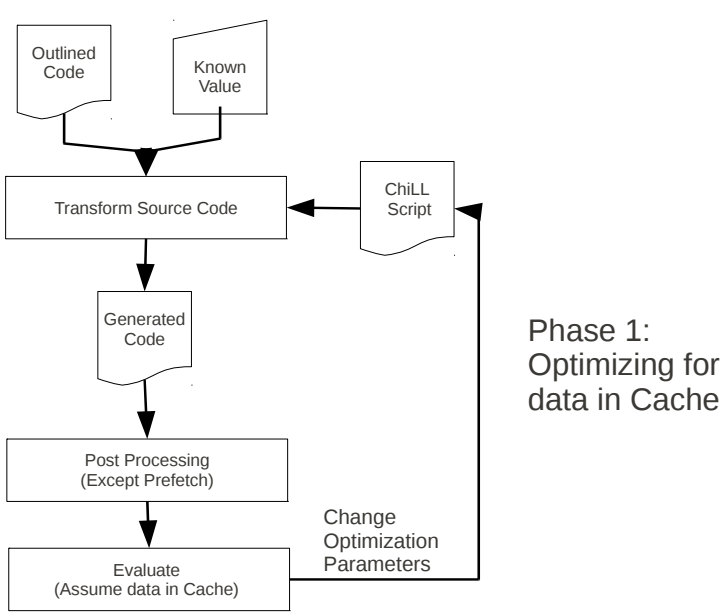

Figure 4.1: Experiment Workflow: Phase 1

known value (Example: number of nonzeros), we exhaustively find the best optimization parameter (Example: unroll factors) for the outlined code. This is performed for all the knowns and all transformation recipes for a function. We perform all the required postprocessing in this phase except for prefetch since we assume that data are in cache.

For each transformation recipe, a specialized implementation of the PETSc function that wraps all the known cases is created in phase 2 (Figure 4.2). The code is then postprocessed for prefetching and evaluated using a driver function. The driver function uses PETSc library calls to read the actual application data and compares the specialized function with the original implementation. The results of this phase are discussed in Section 4.4 .

All experiments were conducted on a single node on the Hopper XE6 system. The best-performing implementation of each function was then integrated into a specialized PETSc library that was linked into the program in the usual way.

To evaluate code generation alternatives on individual kernels, We optimized the functions with two different backend compilers, PGI (version 10.9.0) and Intel compiler (version 12.0.4). Code generated for the Intel compiler was compiled with the -O3 and -fno-alias flag. The -fno-alias informs the compiler to optimize assuming there are no aliases. The corresponding flag for PGI is -Msafeptr. The code for the PGI compiler was optimized

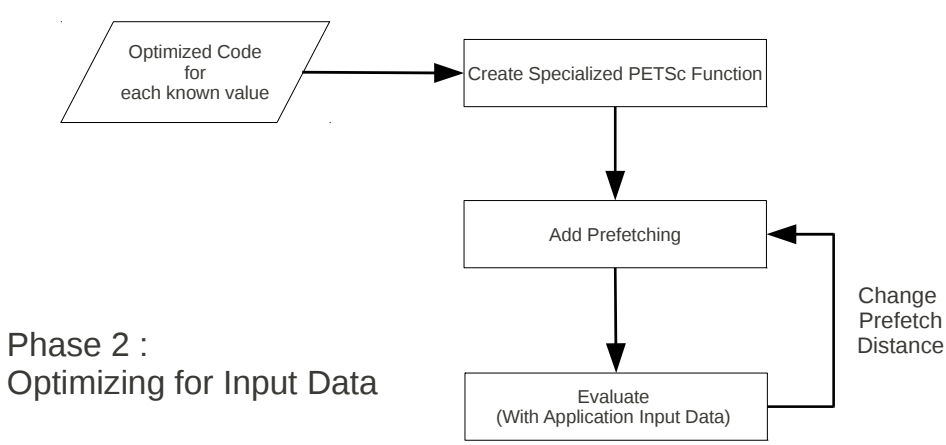

Figure 4.2: Experiment Workflow: Phase 2

with the -fast and -fastsse flags. Table 4.1 shows which compiler and PETSc version were used by the application as provided by the developers. While the optimized functions are compiled by the backend compiler that achieves the best performance, the remainder of the program and the PETSc library are compiled as previously.

4.3 Modification in Application Code

In all three examples, applications had to be recompiled with the modified version of PETSc. PETSc does not directly support the ELL representation but can be simulated by using the AIJ format of matrices. The *MatSetValues* function sets values for multiple rows by accepting inputs for row vectors and the column vectors and their values. PETSc ignores negative column values. Legal column indices with zero in the corresponding column value vector insert a zero in the AIJ format. Using this feature to insert zero values where necessary, each row is guaranteed to contain at least 7 elements.

PETSc has optimized Matrix-Vector Multiplication for the compressed sparse row storage format specifically for banded matrices. It splits the nonzero data into two matrices. The first matrix attempts storing all the diagonal bands while the second matrix stores the rest of the nonzero elements. The mxm matrix is split into blocks of nxn blocks where n is the number of processes. To explain this, we revisit a figure from Chapter 3 displayed in Figure 4.3. Blocks 1,4,8 and 10 are stored in the first matrix while blocks 3,2,6,7 and 9 are stored in the second matrix. Hence, the operation is done in 2 phases. The first phase performs a Matrix-Vector Multiplication for the diagonal blocks (first matrix) and the resultant vector of this operation undergoes a Matrix-Vector Multiplication-Add with the nondiagonal matrix (second matrix). It is important for the application developer to pad only the diagonal matrix to ensure that the input to the Matrix-Vector Multiplication functions has a fixed number of nonzeros per row. The other applications did not require any code modification in their application code.

Table 4.1: Application Setup

Application	PETSc Version	Compiler
PFLOTRAN	PETSc 3.1	PGI 10.9.0
Uintah	PETSc 3.0	gcc 4.6.1
UNIC	PETSc 3.0	Intel 12.0.4

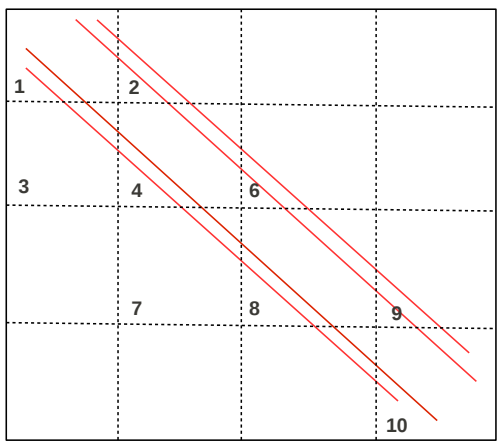

Figure 4.3: Banded Matrices in PETSc

4.4 Impact on Performance of PETSc Functions

We first examine the individual performance of PETSc functions for each application shown in Figures 4.4, 4.5 and 4.6, with measurements for both PGI and Intel compilers. The speedups are compared against a baseline of original PETSc code. In the case of PFLOTRAN, the baseline invokes a separate BLAS `dgemv`, whereas in the other codes, it is represented by C code. We used different optimization strategies for the two compilers because of differences in what code the backend compiler expects to do the best job of code generation. For clarity, the figures show an interesting subset of the experiments we performed.

For PFLOTRAN, we compare not only against the baseline but also a hand-tuned version (first bar) [28]. For some versions, we use the gather version, as shown in Figure 3.3. Loop splitting and a pragma forces SSE code generation on the Intel compiler, as in the example of Figure 3.10 (for more discussion, see Section 4.6). Scalar replacement of arrays reused within the inner loop body is needed for the PGI compiler to place the array elements in registers. Prefetching was used for both compilers. Overall, the best PGI versions obtain speedups of 1.42X and 1.35X on the PETSc functions and employed unrolling, scalar replacement and prefetch and does not perform a gather. The best Intel versions obtain much higher speedups of 1.72X and 1.87X, and use gather, loop splitting and prefetching.

For Uintah, the ELL with the original and gather variant of Figure 3.5 is used for

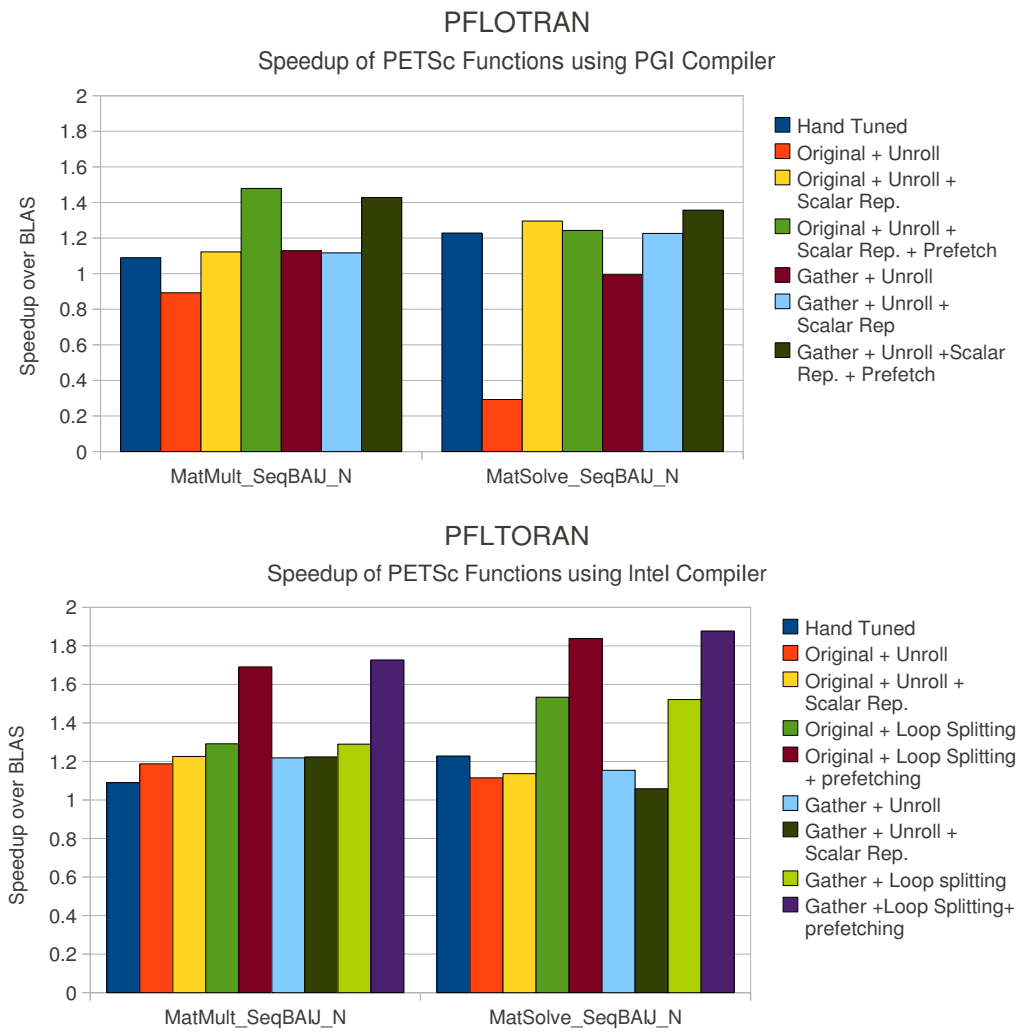

Figure 4.4: PFLOTRAN: Speedup of PETSc Functions

optimization. We use unrolling and prefetching with the PGI compiler and splitting and prefetching with the Intel compiler. Speedup of the best PGI version is 1.36X, while the best Intel performance is 1.48X. For UNIC, the same optimizations and a gather as compared to Uintah are applied, but different ones are profitable. The optimizations applied to Mat Mult had a positive impact on performance in spite of the added control flow to check the number of nonzeros. A snapshot of the specialized code is shown in Figure 3.15. The best PGI versions are just 1.22X and 1.08X faster while the best Intel versions are 1.43X and 1.12X faster.

A summary of the best optimization strategies and their speedup to the orginal code for the PGI and Intel compiler are shown in Tables 4.2 and 4.3.

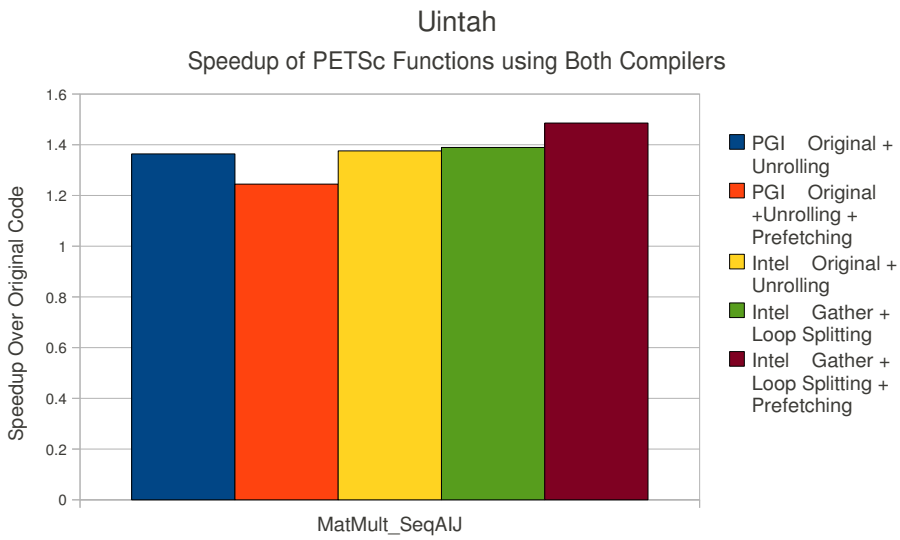

Figure 4.5: Uintah: Speedup of PETSc Functions

4.5 Impact on Performance of Applications

We measured the application using the best-performing versions of the PETSc libraries and compared against the original application. These applications used input data provided by the application developers. In UNIC, we had only one input matrix of a relatively small size. Hence, we were able to scale the problem to only 16 processes while others were scaled to at least 64 processes. Overall, we see application performance gains stable across processes at 1.1X for PFLOTRAN (as compared to an already tuned PETSc and BLAS implementation), a gain of 1.07X to 1.09X on Uintah, and a gain of 1.25X on UNIC (in spite of having unstructured matrices). The results are displayed in Figure 4.7 .

4.6 Impact of Backend Compiler

The performance of the code generated by the PGI compiler is significantly below that of the Intel compiler. In addition, the optimization strategies used for each compiler vary significantly. The gap in performance is especially seen in PFLOTRAN, where unrolling the original code performs much lower than the same optimization applied for the Intel compiler. The performance of the PGI compiler improved substantially by explicitly performing scalar replacement as a postprocess built only for the PFLOTRAN-PETSc routines. Scalar replacement in this function improves instruction scheduling, instruction level parallelism and register reuse. This specific scalar replacement, for register reuse inside inner loop bodies, was not beneficial for the Intel compiler.In fact, CHiLL incorporates a form of scalar replacement in its datacopy, but it is for replacement across loops. In the past, we have

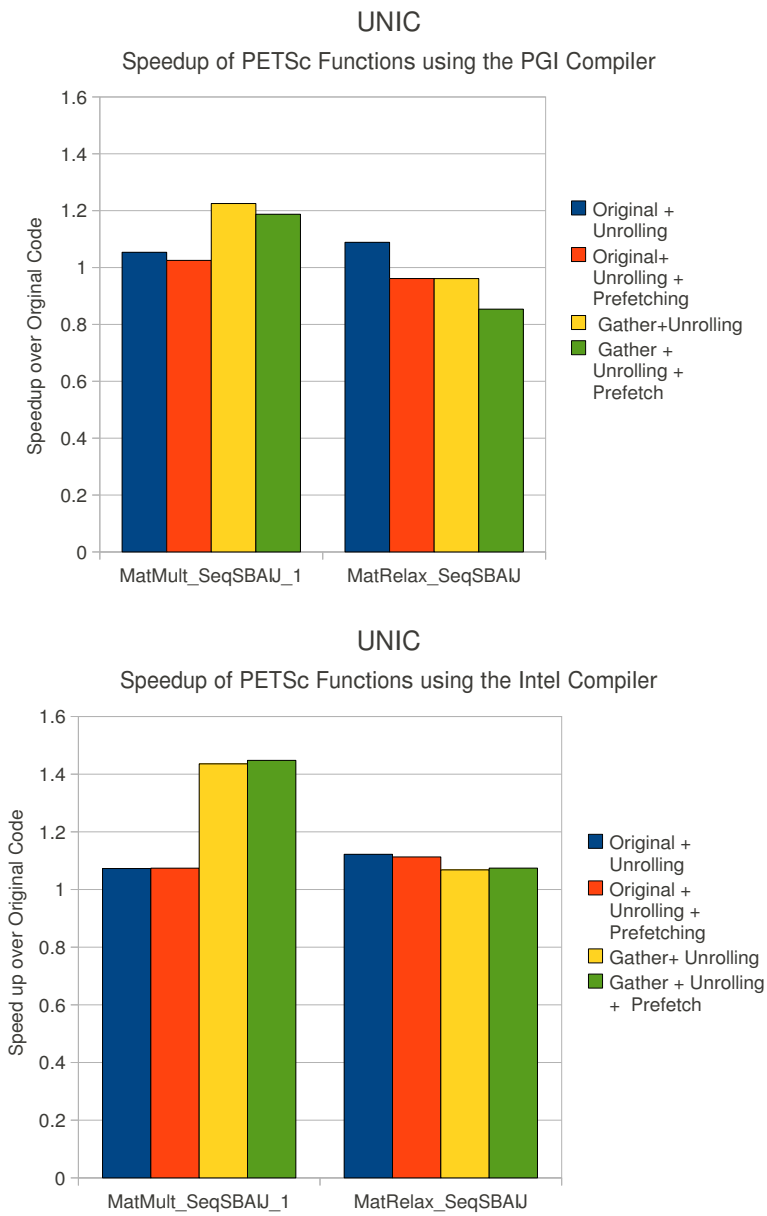

Figure 4.6: UNIC: Speedup of PETSc Functions

found that both the Intel and Nvidia compilers can detect the need for this replacement.

To reduce the effect of cache misses, the PGI compiler generates prefetch instructions for its loops. The generated prefetch instructions may conflict with any explicit calls to the prefetch API. In both Uintah and UNIC, we see a decline in performance when prefetching is performed as a postprocess. However, the Intel compiler does not generate prefetch instructions. Prefetching in PFLOTRAN brought significant gains in both the Intel and PGI compiler, modest gains in Uintah for the Intel compiler and insignificant gains in UNIC.

Table 4.2: Speedup of PETSc Functions using PGI Compiler: Summary

Application	PETSc Function	% Best Performing Optimization	Speedup
PFLOTRAN	MatMult_SeqBAIJ_N	Original + Unrolling + Scalar Rep. + Prefetch	1.42X
	MatSolve_SeqBAIJ_N	Gather + Unrolling + Scalar Rep. + Prefetching	1.35X
Uintah	MatMult_SeqAIJ	Original + Unrolling	1.36X
UNIC	MatMult_SeqSBAIJ_1	Gather + Unrolling	1.22X
	MatRelax_SeqBAIJ	Original + Unrolling	1.08X

Table 4.3: Speedup of PETSc Functions using Intel Compiler: Summary

Application	PETSc Function	% Best Performing Optimization	Speedup
PFLOTRAN	MatMult_SeqBAIJ_N	Gather + Loop Splitting + Prefetching	1.72X
	MatSolve_SeqBAIJ_N	Gather + Loop Splitting + Prefetching	1.87X
Uintah	MatMult_SeqAIJ	Gather + Loop Splitting	1.48X
UNIC	MatMult_SeqSBAIJ_1	Gather + Unrolling + Prefetch	1.43X
	MatRelax_SeqBAIJ	Original + Unrolling	1.12X

The poor performance gains from prefetching in UNIC can be attributed to the unstructured matrix data making it hard to predict a good prefetch distance.

Loop vectorization has very strict requirements that make the generation of SSE code very hard in both compilers. Gather coupled with loop splitting with appropriate postprocessing for the intel compiler exposed SSE instructions in Intel compiler. These versions performed the best across the compilers and functions in which they were used.

4.7 Summary

In this work, we presented a semi-automated framework by which "Joe Programmers" can improve performance significantly by employing transformations scripts designed by "Stephanie Programmers".

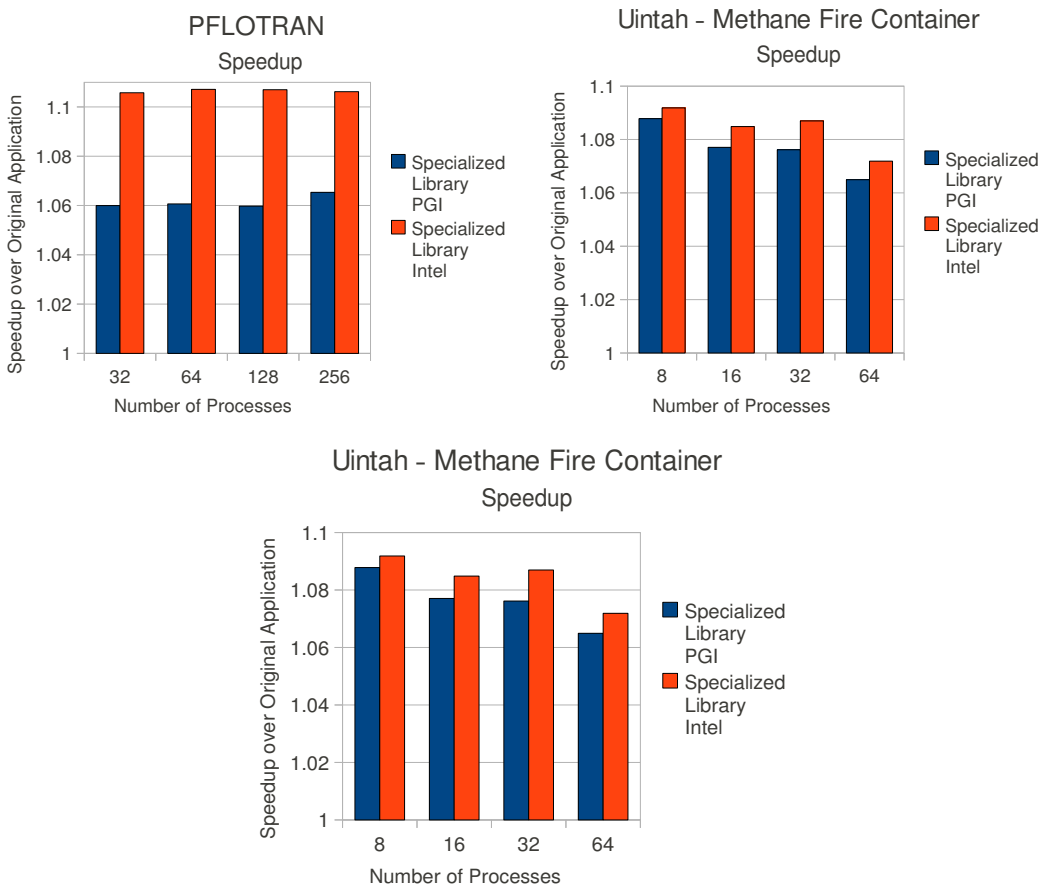

Figure 4.7: Application Speedups

4.7.1 Degree of Automation

There are phases in the workflow that are manually performed in the framework now. Automating some of these would be straightforward, like:

- Profiling data to extract the size of the blocks and a histogram of the nonzeros.
- Postprocessing such as pragmas and prefetching can be incorporated in to the CHiLL framework easily.
- Generating wrapper functions to insert into the code can be done using third party tools like the ROSE compiler.

We feel that two things would require more effort to automate.

- Generating alternate code variants for the gather operations. We believe this can be done by extending the current datacopy command in CHiLL.

- Currently, the exact sequence of CHiLL commands are generated automatically using a python script. We would need to encapsulate these within the compiler framework as a higher-level API for a more robust system. Since we understand what is required now, we can automate this within the CHiLL framework.

4.7.2 Research Contributions

In this work, we improve the performance of sparse computations using nonstandard techniques.

- In PETSc, functions are specialized only for the block size of the data structure. We add one more variable to the specializations, number of nonzero elements calculated at every row. We specialize the inner loops (loop bounds that vary dynamically for every row) for both structured (PFLOTRAN, Uintah) and unstructured (UNIC). The results show strong performance gains in spite of the excessive control flow in the case of unstructured matrices.

- The performance gains in UNIC are strongly attributed to the register level gather operation performed. We showed how to perform a register level gather operation using a source-to-source compiler in tandem with the backend compiler to get the desired effect.

- Finally, we employed the backend compiler to generate SSE instructions on loops that were small and odd bounds. To our best knowledge, using loop splitting along with pragmas to generate SSE instructions is a very rare technique used by maybe only a few programmers.

Apart from the new optimizations, the methodology uses source-to-source compilers along with the backend compilers to perform gather (register) level and generate SIMD instructions that lead to more portable code. In contrast, Williams et al. developed manually tuned libraries with low level SIMD instructions to achieve the same desired effect. Finally, using polyhedral compiler technology on irregular sparse codes is unique in the compiler community.

CHAPTER 5

RELATED WORK AND CONCLUSION

5.1 Related Work

5.1.1 Library-Based Autotuning

Library-based autotuning has been successful for dense and sparse linear algebra [6, 32] and signal processing [10, 23, 31]. OSKI [32] tunes sparse matrix computation automatically. These library-based systems are able to autotune for a particular hardware, but they tune only a fixed set of library kernels and are not able to tune arbitrary computations. With respect to the most closely-related library OSKI, designed for sparse linear algebra, OSKI focuses on tuning for matrix structure and uses a manually-written code generator rather than a compiler to perform its transformations. As compared to our compiler system that can do its work behind the scenes, with a library, the programmer must modify their source code, unless it is embedded inside PETSc and hidden from the programmer. There is a PETSc release that incorporates OSKI, but it would not have helped in the applications in this work:PFLOTRAN's block size was already specified by the programmer as an application feature, and it would not have generated the specialized code for Uintah and UNIC shown in Figure 3.15 .

5.1.2 Compiler-Based Autotuning

Compiler-based autotuning systems can generalize beyond fixed functions. Our own prior work in this area combine compiler models and heuristics with guided empirical evaluations to take advantage of their complementary strengths [8, 7], navigates large search spaces using parallel heuristic search [29, 30], and has developed a unique compiler structure for recipe-based autotuning [7, 14]. Hartono et al. [15] use annotations to describe performance improving code transformations. POET is a scripting language for parameterizing complex code transformations [35]. Pouchet et al. [21, 22] embed legality of affine transformations as linear constraints, thereby combining the code transformation steps and the legality

checking step. Kulkani et al. [18] describe VISTA, which allows selecting the order and scope of optimization phases in the compiler.

5.1.3 Optimizing Linear Algebra

For dense linear algebra, there are several prior techniques to specialize according to matrix size. Herrero and Navarro [16] describe specializing matrix multiplication for small matrices. However, their code variants were generated manually. Gunnels et al. [13] provide strategies for blocking matrices for matrix multiplication at each level of hierarchical memories, but this approach only applies to much larger matrices. Barthou et al. [4] reduce the search space by separating optimizations for in-cache computation kernels from those for memory hierarchy. To generate code variants, they use the X Language controlled by user-provided pragmas. In prior work, we describe a compiler that applies specialization with autotuning for matrix multiply of small rectangular matrices in the context of nek5000, a spectral element code [27, 26]. While the approach for both dense and sparse libraries with small loop bounds rely on many of the same compiler transformations (unroll, specialization with known, SSE), in this work, we employ additional code variants for gather and matrix structure optimization, additional transformations in CHiLL (split, distribution, fusion) to fine-tune code generation, architecture-specific postprocessing (prefetch, pragmas to force SSE code generation) and dynamic code selection in inner loop bodies.

Recent work on benchmarking sparse linear algebra for multicore and many-core architectures employs some of the same optimizations (unroll, block sparse matrices and prefetch), but omits others (gather, ELL representation, dynamic code selection for varying-length nonzero rows). [34]. Significantly, the code is generated by a manually-written script, and the code explicitly contains calls to low-level architecture-specific intrinsics for SSE. Thus, the code is not portable, and it would not be feasible to use such a strategy in the context of specialization. A study of high-performance CUDA implementations was described by [5], but it uses GPU-specific optimizations.

As a design choice, we could have generated SIMD instructions directly from our tool chain using SIMD intrinsics [23, 20, 25]. This would allow us to have finer control over SIMD code generation. Instead,we have the backend compiler perform SIMD parallelization by providing it with dependence and alignment information and forcing the parallelization. In this way, we can rely on the backend compiler's selection of instructions to exploit instruction-level parallelism as well, maintain code portability and simplify the tool chain. Overall, our work is distinguished by its ability to achieve performance gains on existing code using a compiler. In this way, it has minimal impact on the application in which it

is integrated – no modifications to source code or application build process, the ability to specialize beyond what is feasible in a library, and the preservation of high-level code that is translated to tuned code behind the scenes.

5.2 Conclusion

This thesis demonstrated how compiler-assisted specialization could be used to improve the performance of applications that use the PETSc sparse linear algebra libraries. Using the proposed approach, we improved the performance of individual PETSc functions by as much as 1.9X. Overall application performance improved by 9-24% for the three applications we considered. These improvements were obtained over baseline code that is already considered to be tuned and specialized for matrix structure.

How reusable are these specialized libraries? For PFLOTRAN and Uintah, the maximum number of nonzeros and block sizes are fixed by inherit properties of the applications, so for other problem inputs, the same library could be used. This is not the case for the unstructured UNIC, which must test dynamically at each row how many nonzero elements there are. So for UNIC, the tuning would need to be done at run-time, or offline during instantiation of a new problem input. In general, the methodology we employed could be performed for other applications. Over time as we port to new architectures and new generations of backend compilers, most but not all of these optimizations would still be profitable. The transformation recipes and scripts document the optimization strategy for the current execution context and serve as a knowledgeable guide for optimizations for future execution contexts.

REFERENCES

[1] Adhianto, L., Banerjee, S., Fagan, M., Krentel, M., Marin, G., Mellor-Crummey, J., and Tallent, N. R. Hpctoolkit: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience 22*, 6 (2010), 685–701.

[2] Amarasinghe, S., Hall, M., Lethi, R., Pingali, K., Quinla, D., Sarka, V., Shal, J., Luca, R., and Yelic, K. Ascr programming challenges for exascale computing.

[3] Balay, S., Brown, J., Buschelman, K., Eijkhout, V., Gropp, W. D., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 3.1, Argonne National Laboratory, 2010.

[4] Barthou, D., Donadio, S., Duchateau, A., Jalby, W., and Courtois, E. Iterative compilation by exploration of kernel decomposition. In *The 19th International Workshop on Languages and Compilers for Parallel Computing* (2006).

[5] Bell, N., and Garland, M. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009), SC '09.

[6] Bilmes, J., Asanovic, K., Chin, C.-W., and Demmel, J. Optimizing matrix multiply using PHiPAC: A portable, high-performance, ANSI C coding methodology. In *International Conference on Supercomputing* (1997).

[7] Chen, C. *Model-Guided Empirical Optimization for Memory Hierarchy.* PhD thesis, University of Southern California, 2007.

[8] Chen, C., Chame, J., and Hall, M. Combining models and guided empirical search to optimize for multiple levels of the memory hierarchy. In *International Symposium on Code Generation and Optimization* (Mar. 2005).

[9] Davison de St. Germain, J., McCorquodale, J., Parker, S., and Johnson, C. Uintah: a massively parallel problem solving environment. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on* (June 2000).

[10] Frigo, M., and Johnson, S. G. The fastest Fourier transform in the West. Tech. Rep. MIT-LCS-TR728, MIT Lab for Computer Science, 1997.

[11] Guilkey, J. E., and Weiss, J. A. Implicit time integration for the material point method: Quantitative and algorithmic comparisons with the finite element method. *International Journal for Numerical Methods in Engineering* (2003).

[12] Gunnels, J. A., Gustavson, F. G., Henry, G. M., and Geijn, R. A. V. D. FLAME: Formal linear algebra methods environment. *ACM Transactions on Mathematical Software 27*, 4 (2001), 422–455.

[13] Gunnels, J. A., Henry, G. M., and van de Geijn, R. A. High-performance matrix multiplication algorithms for architectures with hierarchical memories. Tech. Rep. CS-TR-01-22, University of Texas at Austin, 2001.

[14] Hall, M. W., Chame, J., Chen, C., Shin, J., and Rudy, G. Transformation recipes for code generation and auto-tuning. In *International Workshop on Languages and Compilers for Parallel Computing* (Oct. 2009).

[15] Hartono, A., Norris, B., and Sadayappan, P. Annotation-based empirical performance tuning using Orio. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)* (Rome, Italy, 2009).

[16] Herrero, J. R., and Navarro, J. J. Improving performance of hypermatrix Cholesky factorization. In *9th International Euro-Par Conference* (2003), pp. 461–469.

[17] Kaushik, D., Smith, M., Wollaber, A., Smith, B., Siegel, A., and Yang, W. S. Enabling high-fidelity neutron transport simulations on petascale architectures. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (2009), SC '09.

[18] Kulkarni, P., Zhao, W., Moon, H., Cho, K., Whalley, D., Davidson, J., Bailey, M., Paek, Y., and Gallivan, K. Finding effective optimization phase sequences. In *ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems* (San Diego, CA, 2003).

[19] Mills, R., Lu, C., Lichtner, P. C., and Hammond, G. E. Simulating subsurface flow and transport on ultrascale computers using pflotran. *J. Phys.: Conf. Ser. 78* (2007).

[20] Nuzman, D., and Zaks, A. Outer-loop vectorization - revisited for short SIMD architectures. In *International Conference on Parallel Architectures and Compilation Techniques* (2008).

[21] Pouchet, L.-N., Bastoul, C., Cohen, A., and Vasilache, N. Iterative optimization in the polyhedral model: Part I, one-dimensional time. In *Fifth International Symposium on Code Generation and Optimization (CGO'07)* (San Jose, CA, 2007).

[22] Pouchet, L.-N., Bastoul, C., Cohen, A., and Vasilache, N. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)* (Tucson, AZ, 2008).

[23] Püschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gačić, A., Voronenko, Y., Chen, K., Johnson, R. W., and Rizzolo, N. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE 93*, 2 (2005), 232–275.

[24] Saad, Y. Sparskit: A basic tool kit for sparse computations, version 2.

[25] Shin, J., Chame, J., and Hall, M. W. Exploiting superword-level locality in multimedia extension architectures. *Journal of Instruction Level Parallelism (JILP) 5* (2003), 1–28.

[26] Shin, J., Hall, M. W., Chame, J., Chen, C., Fischer, P. F., and Hovland, P. D. Speeding up nek5000 with autotuning and specialization. In *Proceedings of the 24th ACM International Conference on Supercomputing* (2010), ICS '10.

[27] Shin, J., Hall, M. W., Chame, J., Chen, C., and Hovland, P. D. Autotuning and specialization: Speeding up matrix multiply for small matrices with compiler technology. In *The Fourth International Workshop on Automatic Performance Tuning* (Oct. 2009).

[28] Sripathi, V., Hammond, G. E., Mahinthakumar, G., Mills, R. T., Worley, P. H., and Lichtner, P. C. Performance analysis and optimization of parallel i/o in a large scale groundwater application on the cray xt5, Nov. 2009.

[29] Tiwari, A., Chen, C., Chame, J., Hall, M., and Hollingsworth, J. K. A scalable autotuning framework for compiler optimization. In *IPDPS* (Rome, Italy, May 2009).

[30] Tiwari, A., Hollingsworth, J. K., Chen, C., Hall, M., Liao, C., Quinlan, D. J., and Chame, J. Auto-tuning full applications: A case study. *International Journal of High Performance Computing Applications* (Aug. 2011), 286–294.

[31] Voronenko, Y., de Mesmay, F., and Püschel, M. Computer generation of general size linear transform libraries. In *International Symposium on Code Generation and Optimization (CGO)* (Seattle, WA, 2009).

[32] Vuduc, R., Demmel, J. W., and Yelick, K. A. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series 16*, 1 (2005), 521–530.

[33] Whaley, R. C., and Dongarra, J. J. Automatically tuned linear algebra software. In *SuperComputing* (1998).

[34] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing 35*, 3 (2009), 178 – 194.

[35] Yi, Q., Seymour, K., You, H., Vuduc, R., and Quinlan, D. POET: Parameterized Optimizations for Empirical Tuning. In *IPDPS* (Long Beach, CA, Mar. 2007).

[36] Yotov, K., Li, X., Ren, G., Garzarán, M. J., Padua, D., Pingali, K., and Stodghill, P. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE 93*, 2 (2005), 358–386.